SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Bioinformatika

PROJEKT

Landau Vishkin Nussinov algoritam

Mauro Barešić

Tin Kovačević

Dino Pačandi

Alen Škvarić

Zagreb, siječnja 2015.

Sadržaj

1. Opis algoritma 1
2. Primjer rada algoritma 4
3. Rezultati testiranja 8
Zaključak 17
Literatura 18

1. Opis algoritma

Glavni problemi s kojima se susreću biolozi prilikom traženja homologa između dviju sekvenci su:

- vremensko trajanje i
- memorijsko zauzeće programa.

Rastom broja nukleotida i proteina koji su mapirani u bazi podataka genoma došlo je i do potrebe za poboljšanjem algoritama koji rade optimalno poravnanje između sekvenci zadanih uzoraka (*engl. patterns*) i dužih tekstova iz baze podataka (*engl. texts*). Optimalno poravnanje podrazumijeva da su uzorak i tekst poravnati tako da se sa što manjim brojem promjena te sekvence mogu izjednačiti. Promjene mogu biti brisanje ili umetanje te zamjena znakova.

Ako kažemo da *m* predstavlja broj znakova u uzorku, a *n* broj znakova u tekstu, onda vrijedi da je $n > m$ (najčešće puno veći). Uvodimo i oznaku *k* za maksimalni broj dozvoljenih razlika između sekvenci. Stariji algoritmi poput *Needleman-Wunsch* ili *Smith–Waterman* rade u $O(m \times n)$ vremenu i zauzimaju $O(m \times n)$ memorije, dok ćemo u nastavku obrazložiti *Landau-Vishkin-Nussinov* algoritam koji radi u $O(k^2 \times n)$ vremenu te zahtijeva $O(m^2+k^2)$ memorije, što predstavlja znatno poboljšanje.

Princip rada prethodno spomenutih starijih algoritama je da se napravi matrica n×m u kojoj se na os apscisa stave svi znakovi iz teksta, a na os ordinata znakovi uzorka te se odrede početne težine znakova. Potom se cijela matrica popunjava redom od gornjeg lijevog kuta s vrijednostima koje se izračunavaju iz prethodno popunjenih susjednih ćelija koje su smještene lijevo, gore i dijagonalno lijevo-gore od promatrane ćelije. Na najveću vrijednost od te tri ćelije se zbraja rezultat usporede znakova uzorka i teksta čiji indeksi odgovaraju promatranom polju u matrici. Tako dobivena vrijednost se zapisuje u promatranu ćeliju, a algoritam se ponavlja do potpunog popunjenja matrice.

Jednom kada je matrica popunjena do kraja, možemo se po njoj kretati te iščitavati koliko bi promjena bilo potrebno da bi izjednačili zadani uzorak i tekst. Smjerovi kretanja po matrici imaju i svoja biološka objašnjenja:

- kretanjem po redu matrice impliciramo da tekst ima određene znakove (nukleotide) koje uzorak nema, odnosno da tekst ima umetnute znakove
- kretanjem po stupcu matrice impliciramo da tekst nema određene znakove, odnosno da su neki znakovi izbačeni (obrisani) iz teksta
- kretanjem po dijagonali impliciramo da je između sekvenci uzorka i teksta došlo do poklapanja, odnosno nepoklapanja između znakova s promatranim indeksima.

Ideja *Landau-Vishkin-Nussinov* algoritma s *k* dozvoljenih razlika je da se ne popunjava cijela n×m matrica već samo vrijednosti koje se nalaze na dijagonalama koje su za najviše *k* udaljene od glavne dijagonale (u oba smjera), jer su upravo te vrijednosti jedine koje su nam bitne i iz njih možemo očitati optimalno poravnanje uz maksimalno *k* razlika. Također, uvodi se i analiza uzorka u matrici veličine m×m iz koje možemo očitati koji znakovi se ponavljaju u uzorku s obzirom na to koliku su razmaknuti, što nam kasnije u algoritmu omogućava bolju iskoristivost pronađenih pravilnosti uzorka u svrhu povećanja efikasnosti cjelokupnog algoritma.

Pseudokod *Landau-Vishkin-Nussinov* algoritma je dan u nastavku, kao i detaljnija objašnjenja oko imena i značenja pojedinih varijabli.

Pseudokod:

```
j := 0;
za i od 0 do n-m+k {
  [1] inicijaliziraj L
  [2] za e od 0 do k {
    za d od -e do e {
      [3] row := max[(L_{d,e-1}+1),(L_{d-1,e-1}),(L_{d+1,e-1}+1)]
      [4.new] dok i+row+d+1 < j {
        [4.new1] uzmi iz S_{i,j} triplet (p,c,f) u kojem je sadržan element iz
            teksta b_{i+row+d+1}
        [4.new2] ako f > 1 {//slučaj a
          [4.new3] ako f != MAXLENGTH(c,row) { //slučaj a1
```

```
            row := row + min(f, MAXLENGTH(c,row))
            idi na 5
          } inače { //slučaj a2
            row := row + f
          }
        } inače { //slučaj b
          [4.new.4] ako b_{i+row+d+1} != r_{row+1} { //slučaj b1
            idi na 5
          } inače { //slučaj b2
            row: = row + 1
          }
        }
      [4.old] dok r_{row+1} = b_{i+row+l+d} {
        row:=row+1
      }
      [5] L_{d,e} := row
      [6] if L_{d,e} = m {
        nađeno podudaranje sa početkom kod b_{i+1}
      }
    }
  }
  [7] ako su novi simboli teksta dosegnuti (j se povećao) rekonstruiraj
novi S_{i,j}
```

Značenje simbola:

m - duljina uzorka

n - duljina teksta

k - maksimalni dozvoljeni broj razlika

R(r) - uzorak

B(b) - tekst

i - trenutni početni indeks u tekstu

j - indeks najdesnijeg simbola iz teksta do kojeg se došlo u prethodnim iteracijama i

d - trenutna dijagonala

e - trenutni broj razlika

maxlength - matrica duljine m x m (kreirana iz uzorka) gdje neki element matrice maxlength[i][j] je ima najduže podudaranje prefiksa ovih sufiksa: R[i,m] i R[j,m]

$L_{d,e}$ - najveći red (*row*) pri čemu vrijedi D(i,j)=e i D(row,j) je na dijagonali d (D je matrica poravnanja)

Sij - sastoji se od liste tripleta T(p, c, f) na temelji oznaka i, j

p - prvi indeks podteksta

c - prvi indeks poduzorka

f – broj poklapanja između podteksta i poduzorka

2. Primjer rada algoritma

uzorak(R): AGACG (m=5)
tekst(B): GGCCGAGCTT (n=10)
k=3

Tablica 1. prikazuje izgled tablice *MAXLENGHT*. Ako promotrimo npr. MAXLENGTH[0][2], vidimo da se uspoređuju nizovi AGACG (kreće od 0) i ACG (kreće od 2). Broj uzastopnih podudaranja iznosi 1, s obzirom na to da je A=A, ali G!=C.

Tablica 1. MAXLENGTH matrica

	A	G	A	C	G
A	5	0	**1**	0	0
G	0	4	0	0	1
A	1	0	3	0	0
C	0	0	0	2	0
G	0	1	0	0	1

Promotrimo što se događa u prvoj iteraciji i (i=0). Početne vrijednosti su sljedeće:
i=0, j=0, S_{ij} = null.
Na tablicama 2. i 3. prikazan je izgled matrice L u prvoj iteraciji i (i=0). Na tablici 2. prikazana je početna vrijednost *rowa* ($L_{d,e}$), tj. *I1*, dok je na tablici 3. prikazan *row*. Retci predstavljaju vrijednost *d*, a stupci *e*. U praksi je matrica L izvedena kao rječnik, s ciljem uštede memorije ili kao polje, s ciljem ubrzanja algoritma. Žute vrijednosti se dobivaju pri inicijalizaciji ([1] u pseudokodu). Te vrijednosti su imaginarne, odnosno odabrane su tako da ne ometaju računanje ostalih vrijednosti $L_{d,e}$. Ako se promotri tablica 3., može se vidjeti koje dijagonale sadrže poklapanje i koliki im je pri tom *k*. U prvoj iteraciji *i* dijagonala 0 ima najbolje preklapanje, tj. najmanji *k* (k=2). Drugim riječima, postoji podudaranje između uzorka i dijela teksta koji počinje od prvog simbola pri čemu je broj razlika jednak 2.

Tablica 2. Matrica L (prikazani l1)

	-2	-1	0	1	2	3
-4					−∞	3
-3				−∞	2	4
-2			−∞	1	3	4
-1		−∞	0	1	3	5
0	−∞	-1	0	1	3	5
1		−∞	-1	0	2	5
2			−∞	-1	0	2
3				−∞	-1	0
4					−∞	-1

Tablica 3. Matrica L, prikazani red(row)

	-2	-1	0	1	2	3
-4					−∞	3
-3				−∞	2	**5**
-2			−∞	1	3	**5**
-1		−∞	0	2	4	**5**
0	−∞	-1	0	2	**5**	5
1		−∞	-1	0	2	**5**
2			−∞	-1	0	2
3				−∞	-1	0
4					−∞	-1

Iako je dobiveno preklapanje za prvu iteraciju *i*, potrebno je još odrediti novi S_{ij}. Njega će se u drugoj iteraciji koristiti za brže pronalaženje podudaranja u slučajevima kada nije dosegnut novi najdesniji simbol ([4.new] u pseudokodu). S obzirom na to da će konačni j biti veći od početnog, koji je 0, prema dijelu pseudokoda [7] kreira se $S_{0,j}$. Slijedi prikaz određivanja novog S_{ij}.

Posebne oznake:

| - odjeljuje stare od novih tripleta

j' – indeks najdesnijeg simbola teksta do kojeg je neka iteracija stigla

(Napomena: kod tripleta (p,c,f) početni indeksi odgovaraju p+1 i c+1, uz pretpostavku da numeriranje indeksa kreće od 1; tripleti s indeksima -1 se mogu zanemariti)

Formiranje novih tripleta:

Ako funkcija *max()* ([3] iz pseudokoda) vrati rješenje dobiveno iz dijagonala d-1 ili d, tada se na staru listu tripleta (određenu u prethodnoj iteraciji ili inicijalizaciji za vraćenu dijagonalu) dodaje triplet ($i+l_1+d-1$, 0, 0). Ako konačan *row* = $L_{d,e}$ bude veći od l_1 dodaje se triplet ($i+l_1+d$, l_1, $L_{d,e}-l_1$).

T(0,0) = |(-1,0,0), l1=0, row=0, j'=0

T(-1,1) = (-1,0,0)|(0,1,1), l1=1, row=2, j'=1
T(0,1) = (-1,0,0)|(0,0,0)(1,1,1), l1=1, row=2, j'=2
T(1,1) = (-1,0,0)|(0,0,0), l1=0, row=0, j'=1

T(-2,2) = (-1,0,0)(0,1,1)|, l1=3, row=3, j'=1
T(-1,2) = (-1,0,0)(0,0,0)|(1,0,0)(2,3,1), l1=3, row=4, j'=3
T(0,2) = (-1,0,0)(0,0,0)(1,1,1)|(2,0,0)(3,3,2), l1=3, row=5, j'=5
T(1,2) = (-1,0,0)(0,0,0)(1,1,1)|(2,0,0), l1=2, row=2, j'=3
T(2,2) = (-1,0,0)(0,0,0)|(1,0,0), l1=0, row=0, j'=2

T(-3,3) = (-1,0,0)(0,1,1)|(1,4,1), l1=4, row=5, j'=2

T(-2,3) = (-1,0,0)(0,1,1)|(1,0,0)(2,3,1), l1=4, row=5, j'=3

T(-1,3) = (-1,0,0)(0,0,0)(1,1,1)(2,0,0)(3,3,2)|, l1=5, row=5, j'=4

T(0,0) = (-1,0,0)(0,0,0)(1,1,1)(2,0,0)(3,3,2)|(4,0,0), l1=5, row=5, j'=5

T(1,3) = (-1,0,0)(0,0,0)(1,1,1)(2,0,0)(3,3,2)|(5,0,0), l1=5, row=5, **j'=6** (j=6)

T(2,3) = (-1,0,0)(0,0,0)(1,1,1)(2,0,0)|(3,0,0), l1=2, row=2, j'=4

T(3,3) = (-1,0,0)(0,0,0)(1,0,0)|(2,0,0), l1=0, row=0, j'=3

$S_{ij} = S_{1,6}$ = (-1,0,0)(0,0,0)(1,1,1)(2,0,0)(3,3,2)(5,0,0)

Tablica 4. prikazuje konačno rješenje za ovaj primjer.

Tablica 4.Konačno rješenje

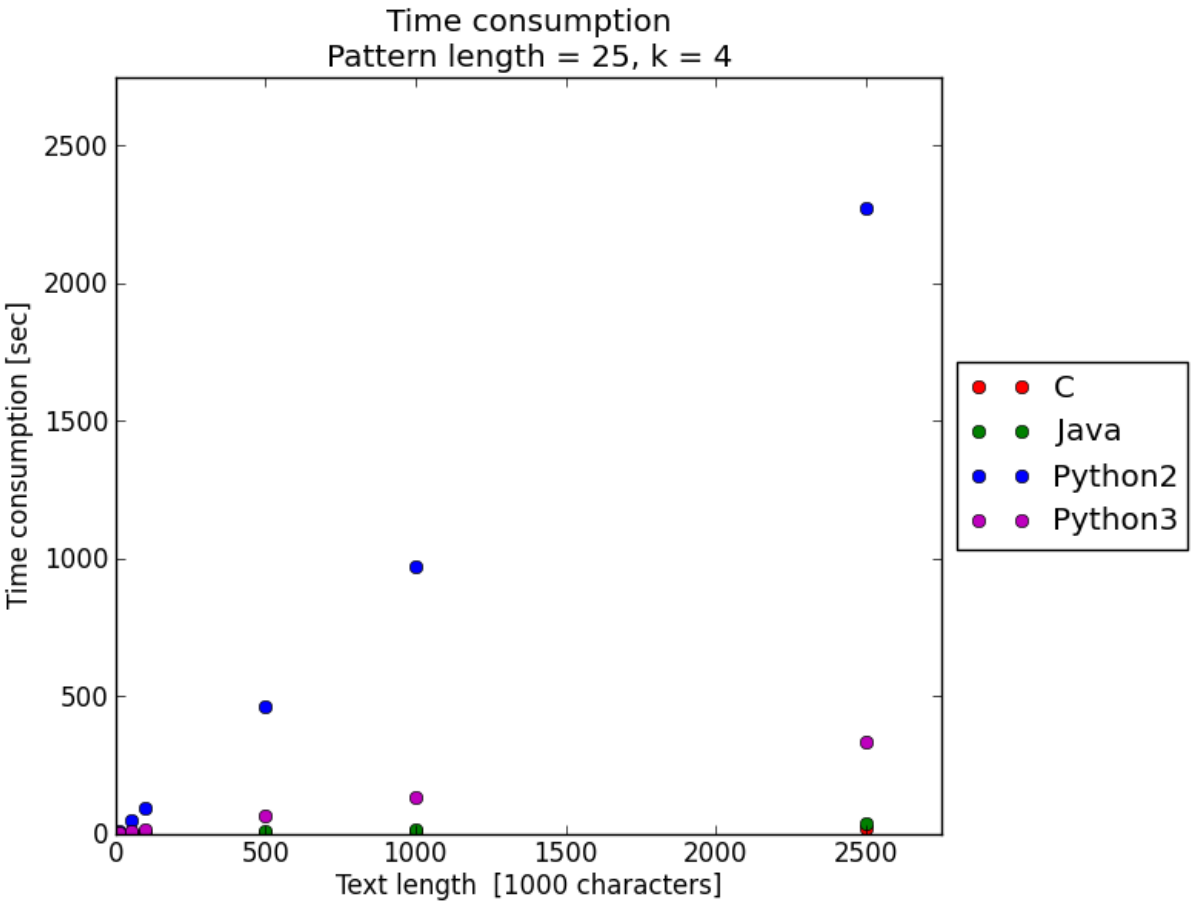
Početni indeks	Krajnji indeks	Broj razlika
0	4	2
1	4	2
2	4	3
3	6	2
4	6	2
5	7	2
6	7	3

3. Rezultati testiranja

U svrhu testiranja performansi implementacija algoritma [1] [2] u različitim programskim jezicima provedeni su eksperimenti na ulaznim podacima različite duljine.

Kao niz znakova korišten je genom bakterije *Escherichia coli* dostupan na web-stranici http://bacteria.ensembl.org/index.html.

Svi eksperimenti provedeni su na računalu s procesorom I*ntel Core i5 M430 2.27GHz* i *4GB radne memorije*.



Slika 3-1 Utrošeno vrijeme obrade ulaznog niza

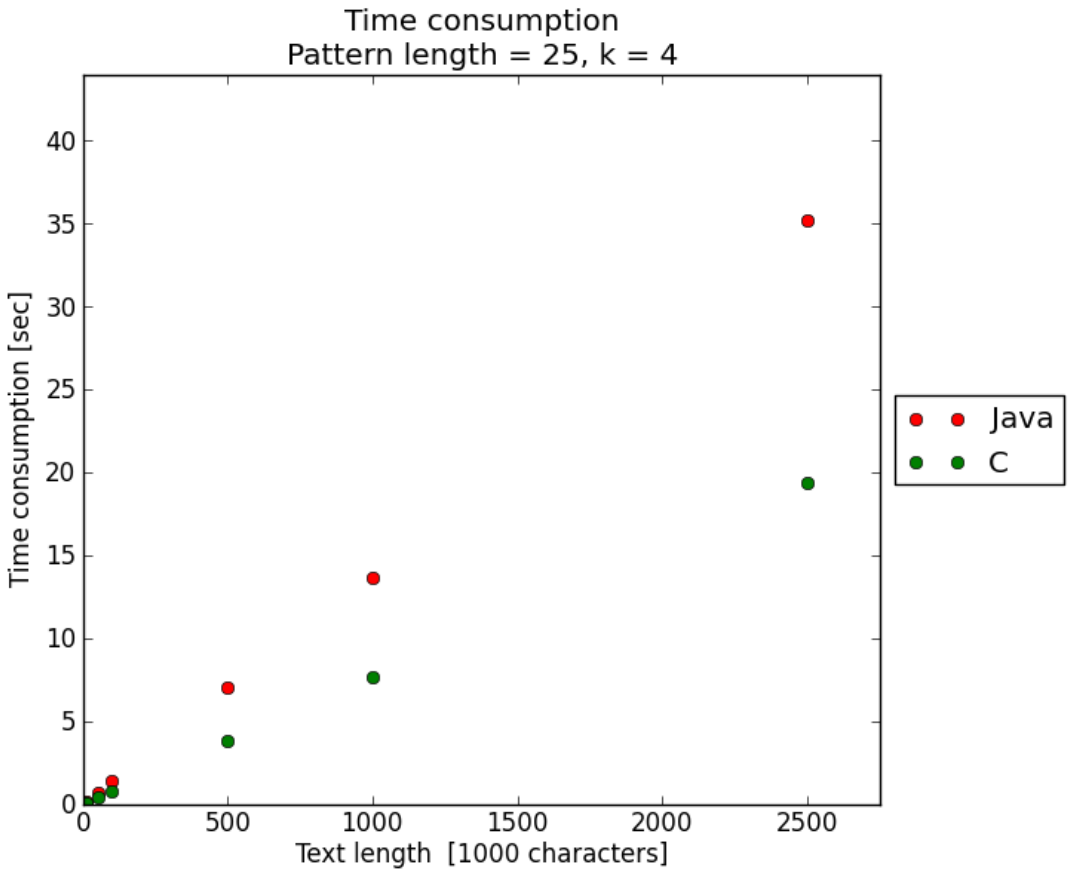
Na slici *3-1* prikazana je potrošnja vremena u ovisnosti o duljini ulaznog teksta. Vrijeme je prikazano u sekundama, a radi lakšeg prikaza kao jedinica duljine teksta

koristi se tisuću znakova. Iz slike je vidljivo da implementacija u Pythonu radi daleko najsporije što je i očekivano. Za obradu ulaznog niza duljine dva i pol milijuna znakova utrošeno je više od 37 minuta, dok je za jednak niz utrošak vremena u C-u oko 20 sekundi. Od svih implementacija, implementacija u C-u se pokazala kao najbrža. Implementacija u Javi radi otprilike duplo sporije od one u C-u što je još relativno brzo; za spomenuti niz utrošeno je oko 35 sekundi.

Tablica 1 – Utrošak vremena [sec] u ovisnosti o programskom jeziku i duljini ulaznog niza

Text length	**C**	**Java**	**Py2**	**Py3**
50	0.001	0.0007	0.032	0.005
250	0.003	0.005	0.265	0.042
1000	0.008	0.041	0.734	0.132
5000	0.039	0.069	4.789	0.671
10000	0.078	0.143	10.062	1.456
50000	0.384	0.699	46.987	6.657
100000	0.769	1.356	93.804	13.309
500000	3.776	7.003	459.842	66.346
1000000	7.6060	13.635	969.417	133.18
2500000	19.394	35.168	2273.813	331.883

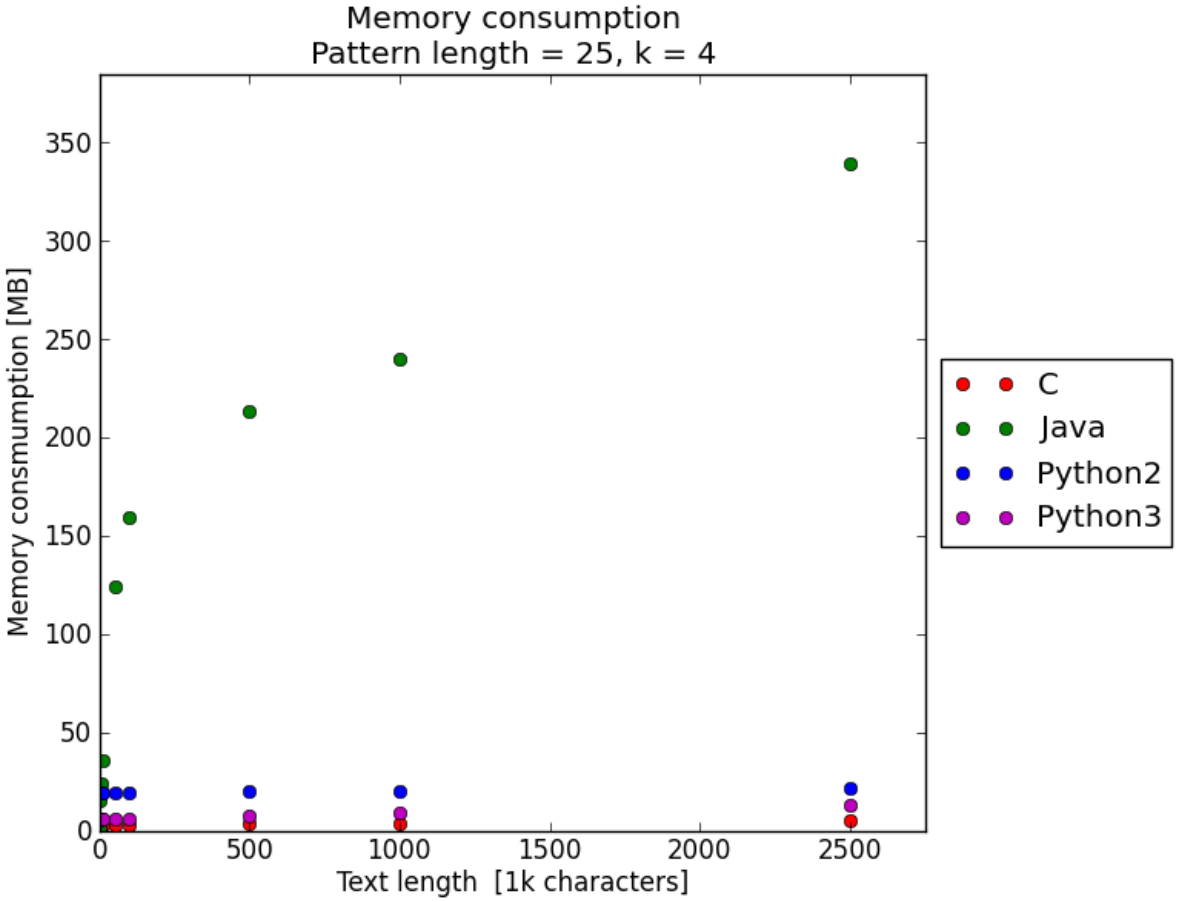
Slika *3-2* prikazuje jednake podatke kao i slika *3-1* samo bez podataka o Pythonu (radi skaliranja prikaza). Vidljiv je linearan rast porasta vremena, jednako kao i za Python implementaciju na slici *3-1.* Ovo je u skladu s očekivanom linearnom ovisnosti vremena obrade o duljini ulaznog teksta.



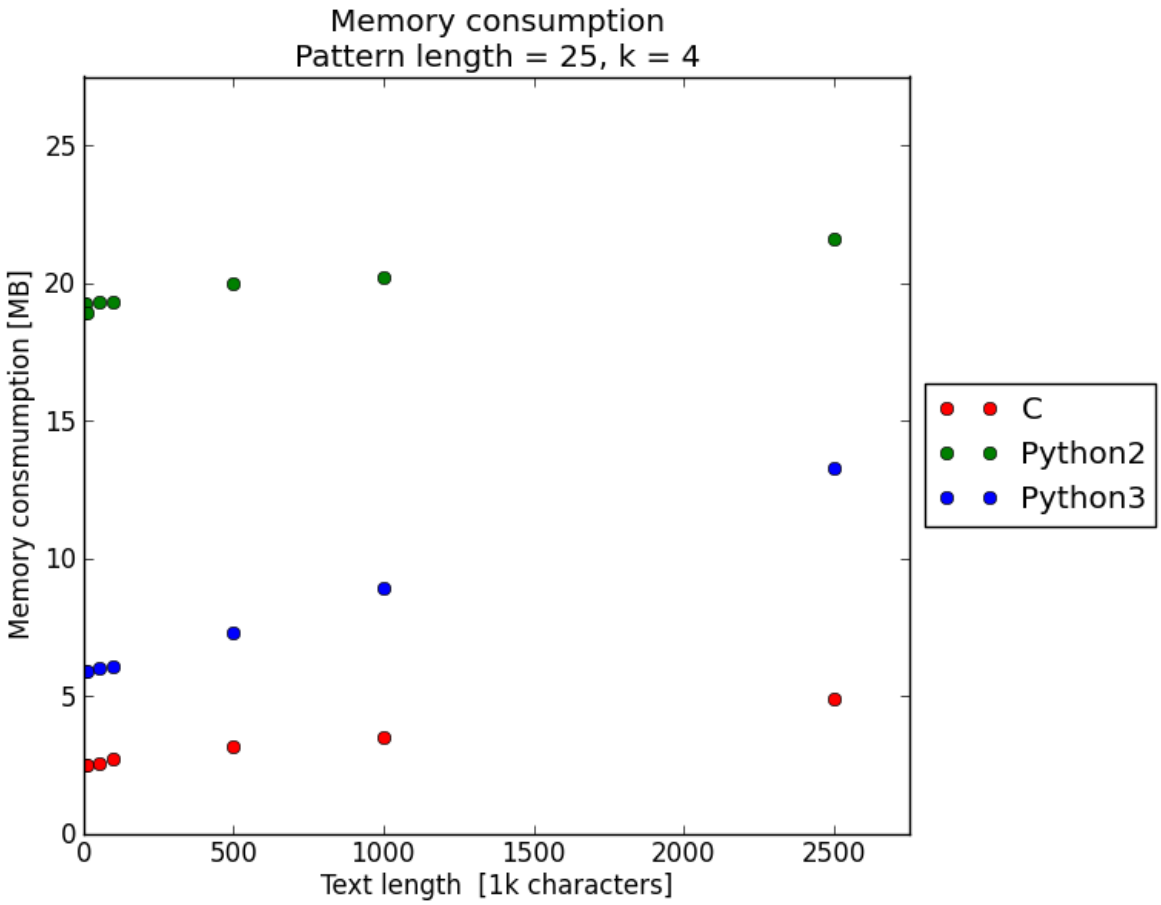
Slika 3-2 Utrošak vremena za C i Java implementaciju

Slika *3-3* prikazuje utrošak memorije u ovisnosti o programskom jeziku i duljini ulaznog niza. Kao i u prethodnom primjeru, duljina teksta izražena je u tisućama znakova. Utrošak memorije izražen je u megabajtima [MB]. Najveću potrošnju memorije ima implementacija u Javi. Za niz najveće duljine potrošnja je čak 350 MB, dok za implementaciju u C-u ne prelazi ni 5 MB. Potrošnja memorije u Pythonu je relativno blizu onoj u C-u. Razlog velike memorijske potrošnje Java implementacije leži u načinu kako se JVM nosi s memorijom. Pošto u Javi nije moguće prisilno osloboditi memoriju, dolazi do nagomilavanja objekata koji se više ne koriste (npr. *liste s tripletima*) što uzrokuje veliku memorijski potrošnju.

Slika *3-4* prikazuje utrošak memorije implementacije u C-u i Pythonu (radi skaliranja slike). Oba porasta su linearna, uz jako malen nagib. Ovdje je vidljivo kako zapravo memorija ne ovisi o duljini ulaznog teksta što je u skladu s teorijom. Malen porast vidljiv na slici može se pripisati samom zauzeću ulaznog niza u memoriji.



Slika 3-3 Utrošak memorije tokom obrade ulaznog niza

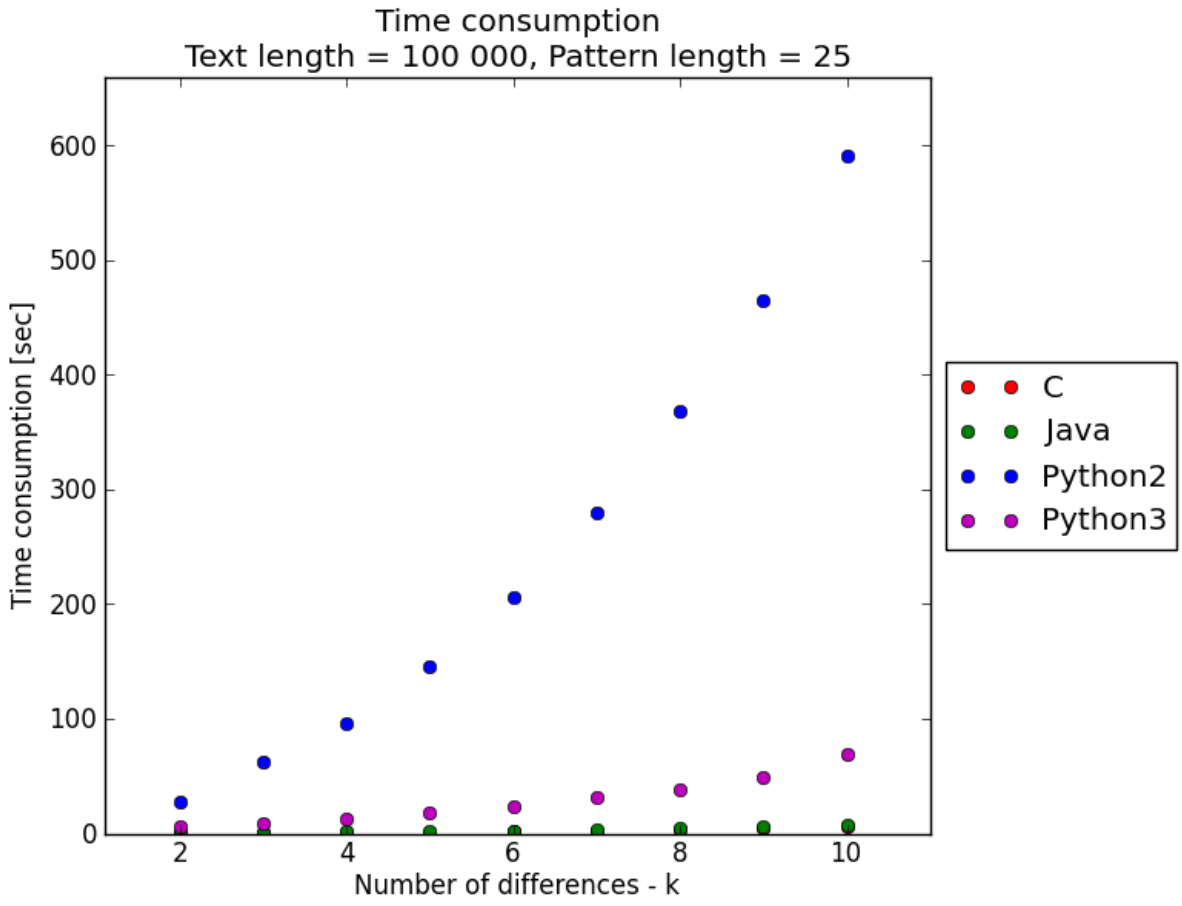


Slika 3-4 Utrošak memorije za C i Python

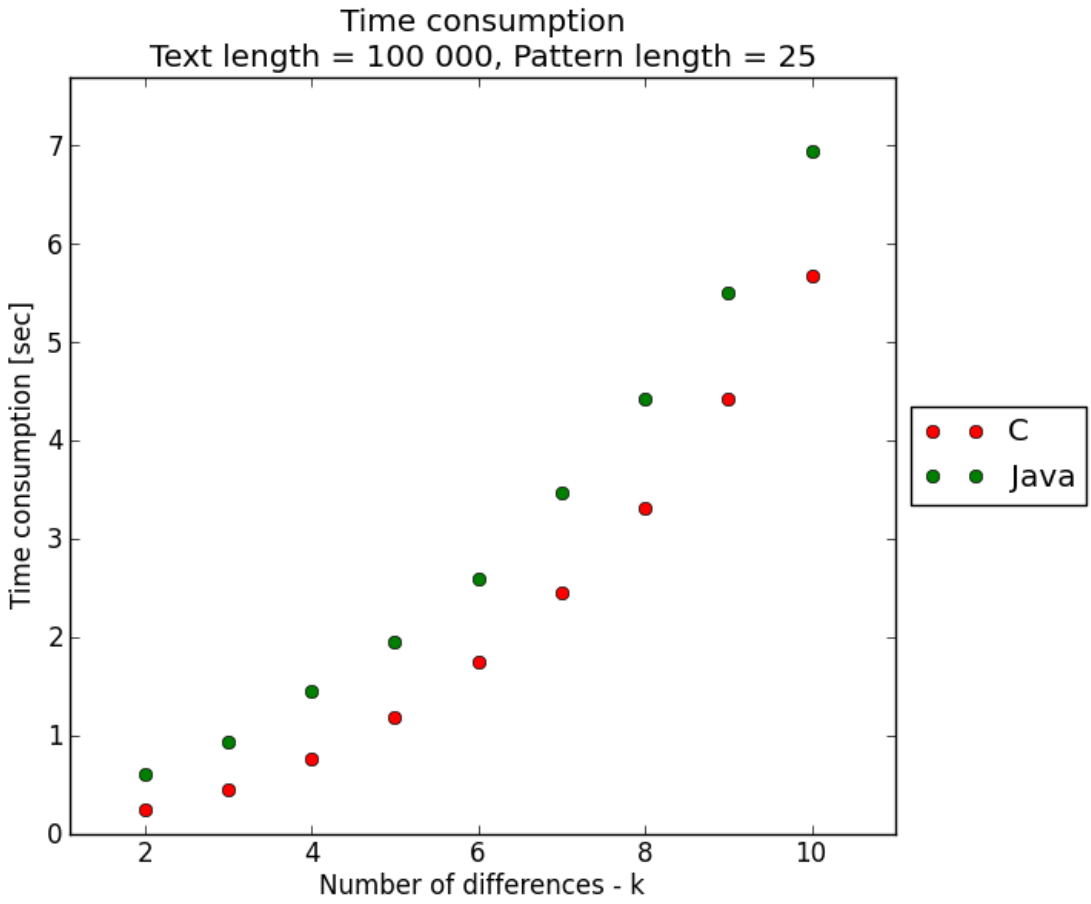
Tablica 2 - Utrošak memorije [MB] u ovisnosti o programskom jeziku i duljini ulaznog niza

Text length	C	Java	Py2	Py3
50	2.476	1.549	19.113	5.924
250	2.480	4.962	19.23	5.594
1000	2.480	15.195	18.902	5.916
5000	2.488	23.518	19.234	5.916
10000	2.5	35.314	18.93	5.924

50000	2.558	124.235	19.293	5.992
100000	2.715	159.582	19.313	6.068
500000	3.168	213.423	20.001	7.3
1000000	3.469	239.976	20.201	8.888
2500000	4.910	338.957	21.625	13.256

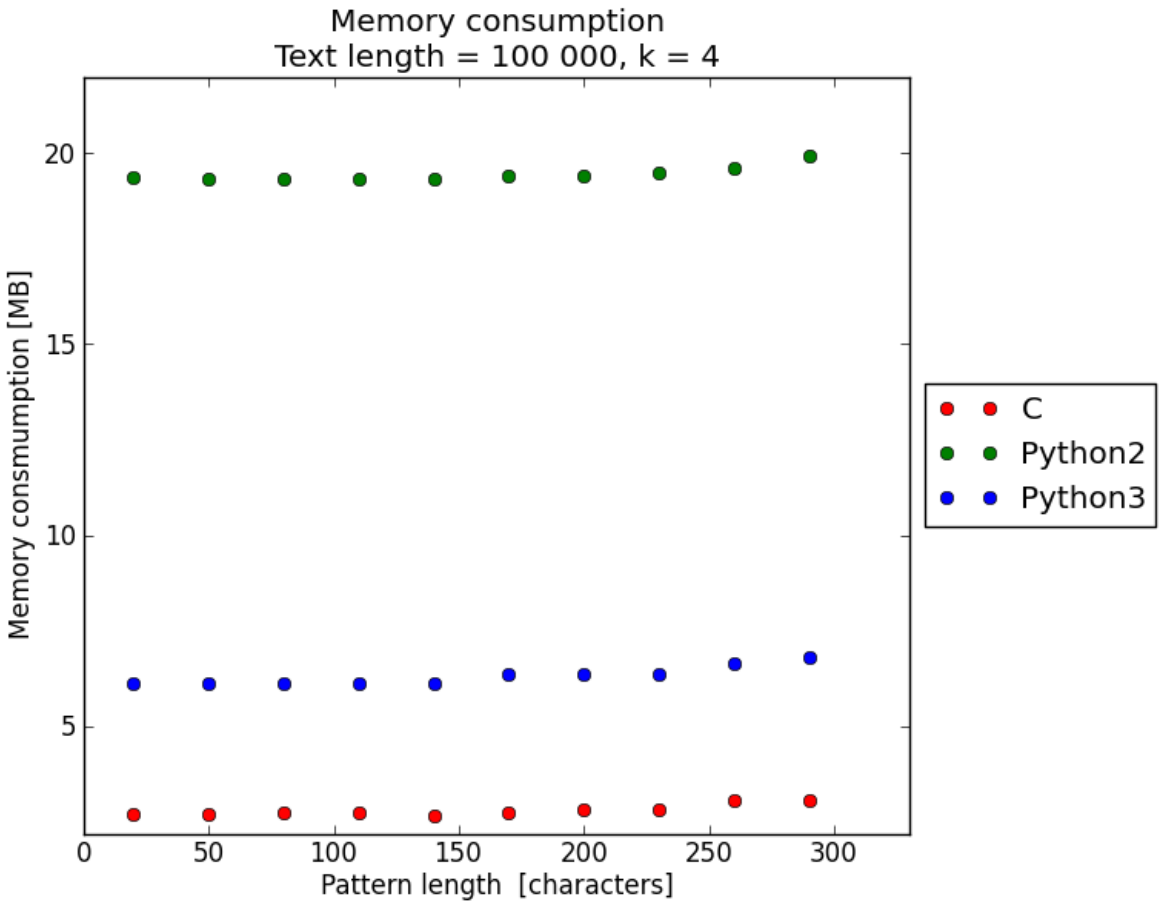


Slika 3-5 – Utrošak vremena ovisno o broju dopuštenih razlika



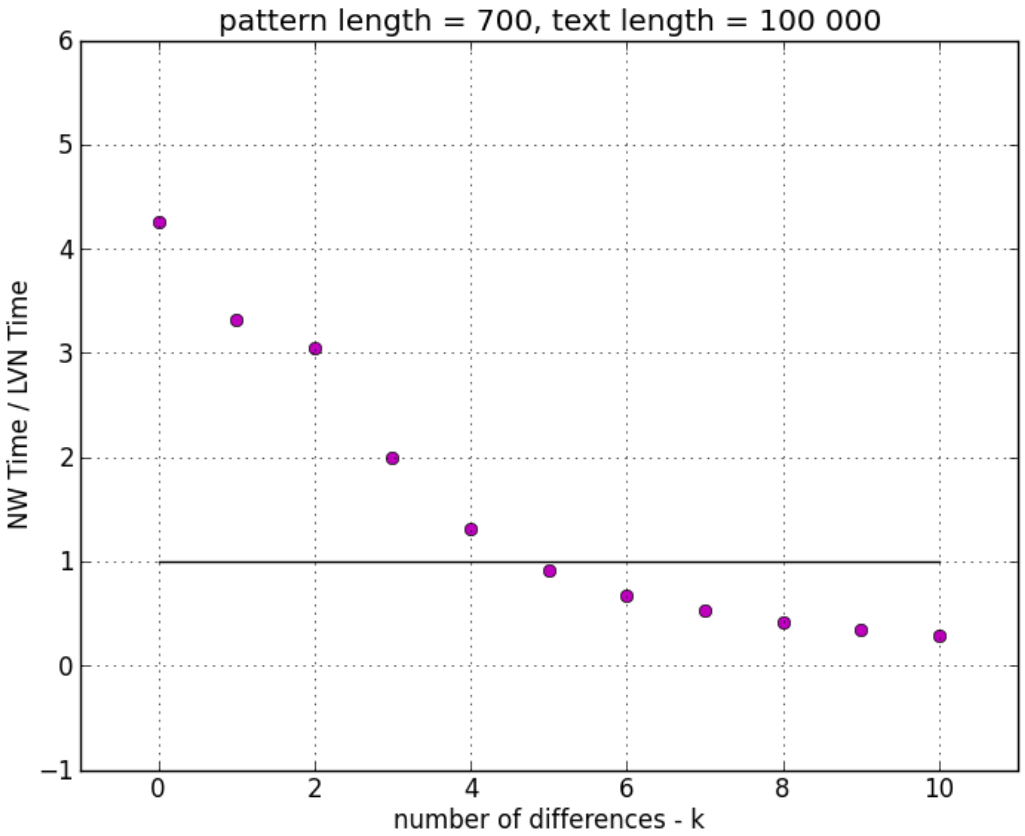
Slika 3-6 - Utrošak vremena za C i Javu ovisno o broju dopuštenih razlika

Slike *3-5* i *3-6* prikazuju utrošak vremena ovisno o broju dopuštenih razlika. Za sve eksperimente koriste se isti tekst i obrazac. Tekst je duljine 100 000 znakova, a obrazac 25 znakova. Iz slika je vidljivo kvadratno povećanje vremena s povećanjem broja dopuštenih razlika što odgovara teoriji.



Slika 3-7 Utrošak memorije u ovisnosti o duljini obrasca

Slika *3-7* prikazuje ovisnost utroška memorije o duljini obrasca. Očekivana teorijska ovisnost je linearna; linearno povećanje utroška memorije s duljinom obrasca. Sa slike se može primjetiti da je porast zanemariv što nije u skladu s očekivanjem (ali nije negativno svojstvo).



Slika 3-8 Omjer brzine izvođenja NW algoritma i LVN algoritma u ovisnosti o k

Slika 3-8 prikazuje omjer brzine izvođenja *Needleman-Wunsch algoritma* i *Landau-Vishkin-Nussinov algoritma* u ovisnosti o broju dozvoljenih razlika (to utječe samo na LVN algoritam). Vidljivo je da LVN algoritam radi brže tako dugo dok *k* ne postane veći od 4. Zanimljivo je da uz *k = 0* (ne dopušta se nikakva razlika) LVN je samo četiri puta brži od NW što je jako malo ako se uzme u obzir da on pretražuje samo glavnu dijagonalu matrice koju NW pretražuje cijelu.

Zaključak

Landau Vishkin Nussinov algoritam nije konstruktivan, dakle ne koristi se za egzaktno utvrđivanje minimalnog broja razlika između dva niza. On je egzistencijalan, tj. služi za utvrđivanje razlikuju li se dva niza u najviše *k* razlika. Ako su dva niza jako slični, tj. *k* je malen, *Landau Vishkin Nussinov* algoritam dobiva dodatno ubrzanje zahvaljujući matrici *MAXLENGTH* koja je rezultat predobrade uzorka. Ako su dva niza jako različiti, algoritam ne dobiva dodatno ubrzanje te je sporiji nego drugi algoritmi polu-globalnog poravnanja. Ukupna vremenska složenost iznosi $O(m^2 + nk^2)$ [1]. Algoritam ima manju memorijsku složenost jer pretražuje samo pojas širine $2k+1$ oko glavne dijagonale matrice poravnanja te mu je ukupna memorijska složenost $O(m^2 + k^2)$ [1].

Literatura

[1] Landau, G. M., Vishkin, U., Nussinov, R.; An efficient string matching algorithm with *k* differences for nucleotide and amino acid sequences; Nucleic acids research, 14(1), 1986, pp: 31-46

[2] Landau, G.M., Vishkin, U.; Fast String Matching with *k* Differences*; JCSS, 37(8), 1988, pp: 63 - 78